Task Scheduling Algorithm for Multicore Processor Systems with Turbo Boost and Hyper-Threading

Yosuke Wakisaka, Naoki Shibata, Keiichi Yasumoto, Minoru Ito
Nara Institute of Science and Technology
Nara, Japan
{*yosuke-w, n-sibata, yasumoto, ito*}*@is.naist.jp*

Junji Kitamichi
The University of Aizu
Fukushima, Japan
kitamiti@u-aizu.ac.jp

***Abstract*—In this paper, we propose a task scheduling algorithm for multiprocessor systems with Turbo Boost and Hyper-Threading technologies. The proposed algorithm minimizes the total computation time taking account of dynamic changes of the processing speed by the two technologies, in addition to the network contention among the processors. We constructed a clock speed model with which the changes of processing speed with Turbo Boost and Hyper-threading can be estimated for various processor usage patterns. We then constructed a new scheduling algorithm that minimizes the total execution time of a task graph considering network contention and the two technologies. We evaluated the proposed algorithm by simulations and experiments with a multi-processor system consisting of 4 PCs. In the experiment, the proposed algorithm produced a schedule that reduces the total execution time by 36% compared to conventional methods which are straightforward extensions of an existing method.**

***Keywords*-Task scheduling algorithm, Multicore, Turbo Boost, Hyper-Threading**

I. Introduction

In recent years, multicore processors have been widely used in various computing environments including data centers and supercomputers. Since the produced heat by the processors is limiting their clock speed, technologies that change clock speed according to the temperature and power consumption of the processor are employed in the latest processors. Such technologies are used in the processors manufactured by Intel and AMD, and they are called Turbo Boost and Turbo Core[1]. We refer to both of the technologies by Turbo Boost, hereafter. Turbo Boost is a technique for increasing the clock speed of some processor cores within the thermal specification when other cores are inactive and the temperature of the processor die is low. Some processors also employ a technology called Hyper-Threading[2] that enables the physical resources of one physical processor to be shared between two or more logical cores to improve the overall throughput.

Task scheduling methods are methods for assigning a series of tasks to a parallel processing system. If we simply apply existing task scheduling methods such as [3], [4], [5] to a system consisting of multicore processors, many tasks are likely to be assigned to a same multicore processor because communication between cores on a same processor die is much faster than communication between dies. In this case, Turbo Boost cannot drastically increase the clock speed of the cores since almost all of the processor cores are active. In some cases, distributing tasks over different dies yields a better schedule because of the boosted clock speed. Thus, we need a scheduling algorithm that takes those technologies into account in order to derive the optimal schedule for systems with these technologies. There is difficulty for some existing scheduling algorithms to consider these technologies, since if tasks are scheduled by those methods that assigns tasks one by one to each processor core, the clock speed for executing the task can be slower than the estimation at the time of assignment, since the clock speed slows down as the subsequent tasks are assigned to the other processors on the same die.

In this paper, we propose a new task scheduling method that takes account of both Turbo Boost and Hyper-Threading technologies and minimizes the processing time. The proposed method takes a task graph specifying dependency among tasks by a directed acyclic graph (DAG) and a processor graph specifying the network topology among available processors, and outputs a schedule which is an assignment of a processor to each task. We constructed a clock speed model for estimating the change of effective processing speed of each core with Turbo Boost and Hyper-Threading. We then constructed a new scheduling algorithm that can more accurately estimate the effective clock speed of each core, utilizing the proposed model.

In order to evaluate the proposed method, we conducted simulations and experiments with actual processors. We compared the proposed algorithm with two algorithms which are extension of the Sinnen's scheduling algorithm[6] that takes account of network contention, and our clock speed model is integrated in a straightforward way. As a result, our method reduced the total processing time by up to 36% in the experiments with a real system. The difference between the scheduled processing time and the actual processing time was 5% in average, and thus we confirmed the task scheduling by our method is effective in the real environments.

II. Related Work

There are many kinds of task scheduling algorithms. In this paper, we assume that task scheduling is assigning a processor to each task, where the dependence of the tasks is represented by a directed acyclic graph(DAG). The problem to find the optimal schedule is NP-hard[6], and there are many heuristic algorithms for the problems.

List scheduling is a classical task scheduling method that assigns the processor that can finish each task to the task in order of a given priority of the tasks. The priority can be given by performing topological sorting on the dependence graph of the tasks, for example. Sinnen et al. extended the classical list scheduling algorithm, and proposed a new method that takes account of the communication delay and network contention[6]. This method assigns the input tasks to the processors while bandwidth in communication paths are reserved for each task so as to minimize the total processing time.

Song et al. proposed a dynamic task scheduling method that executes linear algebraic algorithms on multicore systems with shared or distributed memories. This method scales well, but only applicable to specific tasks.

Jongsoo et al. proposed a task scheduling program called Team scheduling that assigns stream programs to multicore processors [8]. Existing stream programs adjust data transmission timings depending on the data size in the given stream graph so as to efficiently utilize buffers of the processors. This technique is called Amortize. However, deadlock may occur when a large stream graph is input. Team scheduling achieves deadlock-freeness by applying Amortize to a part of the stream graph and suppressing buffer utilization. Moreover, this method achieves better throughput for the same buffer size as the existing methods.

Gotoda et al. proposed a task scheduling method which minimizes recovery time from a single processor failure in multicore processor environments[7]. This method is based on the algorithm [6] proposed by Sinnen et al., and assigns tasks to processors considering both network contentions and recovery time in case of failure of a multicore processor, and produces the optimal task schedule.

As far as we surveyed, there is no existing methods that consider the changes of clock speed by Turbo Boost or Hyper-Threading on a multicore processor system. Unlike these existing methods mentioned above, we propose a new method which targets the environments with a multicore processor system with Turbo Boost and Hyper-Threading. The proposed scheduling method minimizes the total execution time of the input task graph taking account of the two technologies and network contention.

III. Modeling Turbo Boost and Hyper-Threading

In this section, we briefly describe Turbo Boost and Hyper-Threading technologies. The, we describe our model for estimating effective clock speeds determined by the two technologies.

A. Turbo Boost and Hyper-Threading

Turbo Boost is a technology for boosting the clock speed for each core according to the computing load on the processor die. It monitors the temperature and the electric power consumed by the die and dynamically increase the clock speed of some cores if other cores are not used[1]. In this paper, we assume that it determines the clock speed only by the computing load of the all cores on the die.

Hyper Threading is a technology for sharing hardware resources of a physical core among multiple logical cores[2]. When more than one threads are executed on a physical core, the performance of the threads are lower than when only one thread is executed on the physical core. We model this change of execution speed by regarding the clock speed as the index of execution speed at each core, and lowering this speed index of each logical core according to the load on the other logical cores. Hereafter, we call this speed index *effective clock speed*.

B. Modeling

As mentioned above, Turbo Boost and Hyper-Threading technologies can be modeled so that it automatically changes the effective clock speed according to the kind of computational loads on each core. We also assume that the effective clock speed is instantly changed according to the change of core usage, in the course of task execution. It is also assumed that each processor is in one of the following states: (1) idle, (2) computation heavy, (3) memory access heavy, and (4) in-between of (2) and (3).

In order to construct the model, we developed a program that consists of two parts: the part that swaps two randomly selected elements of an 80MB array, and a part that iterates a simple loop staying in the L1 cache. The program repeats executing these two parts in turn. We adjusted the number of loops in the second part of the program, and measured the time to execute this program on multiple cores simultaneously. We specified the processor affinity to each thread so that all threads are executed on the specified cores. We calculated the effective clock speed from the measured processing time.

We used a PC with Intel Core i7 3770T (2.5GHz, 4 physical processors, 8 logical processors, single socket), 16GB memory, Windows 7 (64bit), Java SE (1.6.0 21, 64bit). We used Intel Turbo Boost monitor (Ver2.5) and CPU-Z (Ver1.61.3) to measure the physical clock speed. We first observed how the physical clock speed changes when the number of active physical cores is changed. We show the result of measurement in Table I. The left column shows the processor state, where the 4 pairs represent the usage of four physical processors and each pair like [2, 1] indicates the usage of logical processors within the corresponding

physical processor. The right column shows the clock speed for the corresponding processor usage. The table shows that the clock speed does not depend on the ratio of memory access, but depends only on the number of active physical cores.

In our proposed scheduling method, Hyper-Threading is used only if tasks are already assigned to all physical cores. Thus, we assume that when two logical threads are running on a physical core, the effective clock speed only depends on the ratio of memory access at each logical core. We calculated the effective clock speed from the ratio of execution time by each logical processor to the execution time when one thread is executed on each physical core. The results are shown in Table II.

We constructed a model for effective clock speed from the results above, and we will determine the clock speed from the usage of the processor at which task nodes are assigned using this model.

IV. Problem Formulation

In this section, we formulate the problem of task scheduling taking account of Turbo Boost and Hyper-Threading technologies. The symbols used in this paper is summarized in Table.III.

The task scheduling is to find the schedule S that minimizes the total execution time $lt(S)$ from the given task graph G and processor graph N. A schedule is a tuple of an assignment of a processor to each task, the starting and finishing time of each task node, and the information of bandwidth reservation on each processor link.

A task graph G is a DAG in which each node represents a task to be performed. Each node in a task graph is called a task node. The amount of computation to finish task node v is denoted by $C_{comp}(v)$. A directed arc in the graph is called a task link, and a task link from node v_a to v_b indicates that task v_a must be completed before task v_b begins. A task link e also represents communication between two nodes, and the amount of data transfer for this link is denoted $C_{comm}(e)$. The set of all task nodes and the set of all task links are denoted **V** and **E**, respectively. Fig. 1 shows an example of a task graph consisting of 3 task nodes and 2 task links.

Table I: Effective clock speed with Turbo Boost

Processor states	Effective clock speed
[2, 1], [1, 1], [1, 1], [1, 1]	3.7
[2, 1], [2, 1], [1, 1], [1, 1]	3.5
[2, 1], [2, 1], [2, 1], [1, 1]	3.3
[2, 1], [2, 1], [2, 1], [2, 1]	3.1
[3, 1], [1, 1], [1, 1], [1, 1]	3.7
[3, 1], [3, 1], [1, 1], [1, 1]	3.5
[3, 1], [3, 1], [3, 1], [1, 1]	3.3
[3, 1], [3, 1], [3, 1], [3, 1]	3.1
[4, 1], [1, 1], [1, 1], [1, 1]	3.7
[4, 1], [4, 1], [1, 1], [1, 1]	3.5
[4, 1], [4, 1], [4, 1], [1, 1]	3.3
[4, 1], [4, 1], [4, 1], [4, 1]	3.1
[1, 1], [1, 1], [1, 1], [1, 1]	2.5
[2, 2], [2, 2], [2, 2], [2, 2]	2.6
[3, 3], [3, 3], [3, 3] [3, 3]	2.3
[4, 4], [4, 4], [4, 4], [4, 4]	2.5

Processor state: 1:idle, 2:computation heavy, 3:memory access heavy, 4:in-between of 2 and 3

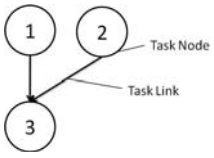


Figure 1: Example task graph

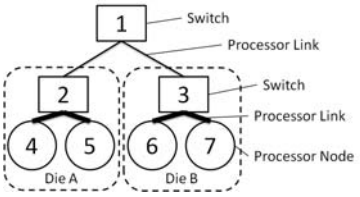


Figure 2: Example processor graph

A processor graph is a graph that represents the network topology between processors. A node with only one link is called a processor node, that corresponds to one processor core. A node with two or more links is called a switch. A switch is not capable of executing a task but only relays communication. An edge is called a processor link, and it represents a bidirectional communication link between processors and switches. One multicore processor is represented by multiple processor nodes, a switch and processor links connecting them. The set of all processor nodes and the set of all processor links are denoted **P** and **R**, respectively. $freq(p, s)$ denotes a function that gives the effective clock speed of processor p from the state s of all cores on the same die. Fig. 2 shows an example of a processor graph consisting of three processor cores and three switches, or two multicore processors and a switch.

In this paper, we use a network contention model based on the model proposed by Sinnen et al.[6], and we make the following assumptions. When data transfer is performed over network links between two processor nodes, due to bandwidth limitation these network links cannot perform

Table II: Effective clock speed with Hyper-Threading

Processor states	Ratio of exec. times	Effective clock speed
[1 , 1]	1.0	2.5
[2 , 2]	0.84	2.6
[3 , 3]	0.76	2.3
[4 , 4]	0.79	2.5

Table III: Symbols used in this paper

Symbol	Meaning
$\mathbf{V}$	Set of all task nodes
$\mathbf{E}$	Set os all task links
$\mathbf{P}$	Set of all processor nodes
$\mathbf{R}$	Set of all processor links
$lt(S)$	Completion time of the last task node in schedule S
G	Task graph
N	Processor graph
$C_{comp}(v)$	Computation cost for task node $v \in V$
$C_{comm}(e)$	Communication cost for task link $e \in E$
$freq(s)$	Effective clock speed determined from processor state s
n_i	Task node for the i-th task
$w(v)$	Execution time for task node v
$c(e)$	Communication time at task link e
$proc(n)$	Processor assigned to task node $n \in V$
$pred(n_i)$	Set of all parent nodes of n_i

other data transfers. We also assume the following conditions are satisfied: if data are transferred through a series of processor links, downstream links cannot start data transfer before upstream links; communication inside a same die finishes instantly; all processors on a same die share a network interface that can be used to communicate with devices outside the die; all communication links outside dies have the same bandwidth. Data transfer correnspounding to task link e over a communication link outside dies requires $C_{comm}(e)$ length of time. One processor can execute only one task at a time. A task node cannot be executed until all execution of parent nodes and all corresponding data transfers are finished. It takes $C_{comp}(v)/freq(p,s)$ length of time for processor node p to finish execution of task node v, where s is the state of all cores on the same die as p.

V. Proposed Algorithm

In this section, we explain our scheduling algorithm. This scheduling problem is known as NP-Hard[6], and thus we propose a heuristic algorithm considering both network contention and change of clock speed with Turbo Boost and Hyper-Threading technologies by extending the scheduling algorithm proposed by Sinnen et al.[6]. We use the clock speed model described in Section 3 for this purpose.

Algorithm 1 List scheduling

INPUT: Task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and processor graph $H = (\mathbf{P}, \mathbf{R})$.

1: Sort nodes $n \in V$ into list L, according to priority scheme and precedence constraints.
2: **for** each $n \in L$ do **do**
3: Find processor $p \in \mathbf{P}$ that allows earliest finish time of n.
4: Schedule n on p.
5: **end for**

Algorithm 2 Scheduling considering network contention

INPUT: Task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and processor graph $H = (\mathbf{P}, \mathbf{R})$.

1: Sort nodes $n_j \in V$ into list L in descending order of bl, according to precedence constraints.
2: **for** each $n \in L$ do **do**
3: Find processor $p \in \mathbf{P}$ that allows earliest finish time of n_j, taking account of network bandwidth usage.
4: **for** each $n_i \in pred(n_j)$ in a definite order **do**
5: **if** $proc(n_i) \neq p$ then **then**
6: determine route $R = [L_1, L_2, ..., L_l]$ from $proc(n_i)$ to p.
7: schedule e_{ij} on R.
8: **end if**
9: **end for**
10: schedule n_j on p.
11: **end for**
12: **return** the schedule.

Scheduling algorithms based on the list scheduling do not perform well with systems where clock speeds of the processors are controlled by Turbo Boost or Hyper-Threading. This is because the list scheduling assigns a processor to each task node in turn, and it cannot know the effective clock speed for each task during assignment, since the effective clock speed is influenced by the execution of succeeding tasks. The proposed method tentatively assigns processors to the all succeeding tasks assuming that these succeeding tasks are executed in a predetermined fixed clock speed. Then, it estimates the execution time of the tasks by applying the proposed model for the effective clock speed. Although this execution time is calculated using the tentative schedule, we regard this execution time as an approximation of the actual execution time and make the schedule based on it.

Hereafter, we first explain the traditional list scheduling algorithm, followed by the extension by Sinnen et al. for considering network contention. Then, we give the details of the proposed algorithm.

A. Existing Algorithms

The classical list scheduling algorithm is shown in Algorithm 1. In the list scheduling, each task is assigned to the processor that allows the earliest finish time of the task, in descending order of bl, that is the length of remaining schedule.

The algorithm proposed by Sinnen, et al. is shown in Algorithm 2. Below, we give explanation for the pseudocode.

Line 2 to 11: Each task node $n_j \in L$ is assigned a processor in order of the position in L.
Line 3: The processor assigned to n_j is determined taking account of network bandwidth usage. Reserved bandwidth in line 7 is referred here.

Algorithm 3 The proposed scheduling algorithm

INPUT: Task graph $G = (V, E, v_{start}, C_{comp}, C_{comm})$, processor graph $N = (P, R)$ and frequency model $freq$

1: $S_{prev} =$ an empty schedule
2: Sort nodes in V into list L in descending order of the length of succeeding tasks, according to precedence constraints.
3: **for** $n_i \in L$ n_i is the first element in L **do**
4: $S_{cur} =$ an empty schedule, $T_{cur} = \infty$
5: **for** each $p_i \in P$ **do**
6: $S_{cand} = S_{prev}$
7: **for** each preceding task n_j of n_i **do**
8: **if** p_i is not assigned to n_j on S_{cand} **then**
9: Determine route $r = [L_1, L_2, ..., L_l]$ from the processor assigned to n_j to p_i
10: Reserve bandwidth C_{comm}(the task link from n_j to n_i) on route r in S_{cand}
11: **end if**
12: **end for**
13: Calculate finishing time of n_i including communication time assuming that n_i is executed on p_i with the fixed clock speed, and add the information of finishing time to S_{cand}
14: Schedule all unassigned tasks in S_{cand} using Algorithm 2 and substitute the resulting schedule for S'_{cand}
15: Calculate execution time of each task node in S'_{cand} with the proposed model for effective clock speed
16: **if** the total execution time of $S'_{cand} < T_{cur}$ **then**
17: $S_{cur} = S_{cand}, T_{cur} =$ the total execution time of S'_{cand}
18: **end if**
19: **end for**
20: Remove n_i from L
21: $S_{prev} = S_{cur}$
22: **end for**
23: **return** S_{cur}

Line 4 to 9: Bandwidth of e_{ij} is reserved for the network route between the processor assigned to n_i (which is the parent node of n_j) to the processor assigned to n_j.

B. Scheduling Considering Frequency Change

The pseudo code for the proposed algorithm is shown in Algorithm 3. In the algorithm, S_{prev}, S_{cur} and S_{cand} retain portions of schedules in which only a part of the all assignment is specified. The total execution time for these incomplete schedules can be calculated by assigning processors to the all unassigned tasks using algorithm 2, and then applying our clock speed model. S_{prev} retains the best incomplete schedule in which the all tasks prior to n_i are assigned, and other tasks are not assigned. S_{cur} and T_{cur} retain the current best incomplete schedule in which n_i and the prior tasks are assigned, and the corresponding execution time, respectively. Below, we give explanation for the pseudocode.

Line 3 to 22: A processor is assigned to each task node.
Line 5 to 19: Each processor p_i is tentatively assigned to the first task n_i in list L so that the processor that achieves the earliest finish time of the all tasks is found.
Line 7 to 13: Processor p_i is assigned to task link n_i.
Line 14: The all succeeding tasks after n_i are scheduled assuming that they are executed in a fixed clock speed.
Line 15: The total execution time for this schedule is calculated using the proposed clock speed model.
Line 16 to 18: The best processor to be assigned to n_i is determined by the execution time.

VI. Evaluation

In order to evaluate the efficiency of the schedule generated by the proposed method and the accuracy of the proposed model for effective clock speed, we conducted experiments using a real system and simulation-based comparisons.

A. Compared Methods

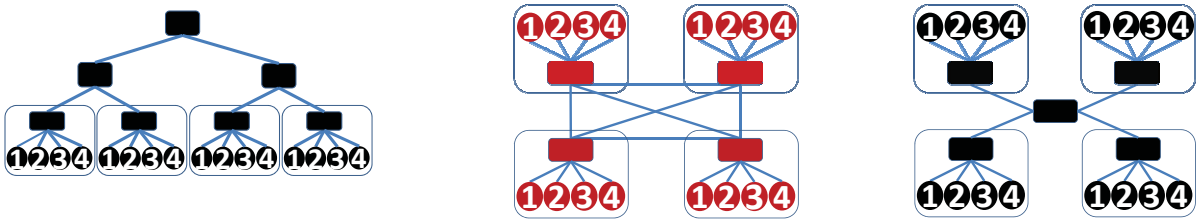
As we described in Section 2, we could not find an existing task scheduling method considering Turbo Boost or Hyper-Threading. In order to make fair comparisons, we integrated our clock speed model into the Sinnen's scheduling algorithm in a straightforward way and made two methods: **SinnenPhysical** that is a scheduling algorithm that tries to assign only physical processors to the tasks, and **SinnenLogical** that tries to assign all logical processors to the tasks. These two methods are extended so that they utilize the clock speed model when choosing the best processor that allows the earliest finishing time of each task[1].

As a preliminary experiment, we compared **SinnenPhysical** and **SinnenLogical** with the original method proposed by Sinnen et al. that does not consider the changes of clock speed at all, and confirmed that **SinnenPhysical** and **SinnenLogical** generate better schedules than the original algorithm for our system configuration.

B. Configuration

We used a PC with Intel Core i7 3770T (2.5GHz, 4 physical processors, 8 logical processors, single socket), 16GB memory, Windows 7 (64bit), and Intel 82579V Gigabit Ethernet Controller as a computing node. The system consists of four of these PCs connected with Gigabit Ethernet. We implemented the programs to execute the scheduled tasks using the standard TCP socket with Java SE (1.6.0 21, 64bit). In order to eliminate the influence of the operating system, we stopped the background tasks except the ones required for continuing the minimum operations of the OS. We set the threads' affinities to each of processor cores so that each task node is executed on the core specified by the schedule. We tested the two real network topologies shown in Fig. 3. For the simulation, we also tested a fully-connected network topology. We used 420Mbps as the bandwidth of processor

[1]At Line 3 in Algorithm 2, the processor assigned to n_j is determined taking account of the two technologies. Only already assigned tasks are considered to estimate the effective clock speed.



Tree-shaped topology Fully-connected topology Star-shaped topology

Figure 3: Processor graphs used in evaluation

links outside the dies, that is obtained by measuring the network bandwidth on the above system.

We used task graphs for Robot Control and Sparse Matrix Solver from the Standard Task Graph Set[9], [10] in our evaluation. The Sparse Matrix Solver has 98 nodes and 177 links and represents a sparse matrix solver of an electronic circuit simulation generated by the OSCAR FORTRAN compiler. This graph has relatively high level of parallelism. The Robot Control has 90 nodes and 135 links. The Robot Control task graph represents a parallel task for Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator. The Robot Control task has lower level of parallelism compared to the Sparse Matrix Solver. Since the ratio of computation and memory access is not specified in these task graphs, we used the 4th state of the processor load, which is in-between of computation heavy and memory-access heavy states described in Section 3, for the all task nodes.

C. Efficiency of Generated Schedules

We evaluated the efficiency of generated schedules by comparing the generated schedules with the proposed method and the two comparison methods. We calculated the execution time of generated schedules with simulation, and measured the execution time on the real system by assigning and executing tasks on the processors in the real system. We performed simulations with the combinations of the two task graphs and the three processor graphs. In the experiments, we combined the two task graphs and the two processor graphs except the fully-connected topology.

We compared the total execution time of the schedules generated by the proposed method to the schedules generated by the compared methods. The simulation results and the experimental results are shown in Fig. 4 and 5, respectively. These results show that the proposed method reduced the total execution time by up to 43% in the simulation, and up to 36% with the real system. We can see that the proposed method has greater effect on the Sparse Matrix Solver task than on the Robot Control task. This is because the Robot Control task has less parallelism, and this limits the freedom for scheduler to choose a processor for each task. Thus, the algorithm has smaller freedom for controlling the generated schedule. The results also show that our method has greater effect on the tree-shaped or the star-shaped network topology than the fully-connected topology. This is because the fully-connected topology requires less communication time than other two toplogies.

D. Accuracy of Effective Clock Speed Model

In order to evaluate the accuracy of the proposed model for effective clock speed, we compared the total execution time of the task graphs on the real system with the simulated results. Fig. 6 and 7 show the results. In the experiment, the error of the estimated execution time was no more than 7%, and the average error was 4% . We also chose 20 random task nodes from the graphs and compared the distribution of the execution time for each of the nodes with the simulated results. Fig. 7 shows the 90%-tiles of the measured execution time with the simulated time. The maximum error was 16% and the average error was 8.5%.

The difference between the results in the simulation and the experiments is probably coming from the fluctuation of network bandwidth and the processor load by the background tasks in the OS. However, the errors in the results are not significant, and the proposed clock speed model is sufficient for estimating the execution time of each task with Turbo Boost and Hyper-Threading.

VII. Conclusion

In this paper, we formulated the problem for generating task schedules minimizing the total execution time of task graphs considering network contention and multicore processors with Turbo Boost and Hyper-Threading technologies. We also modeled the two technologies so that the effective processing speed of each core can be estimated. Then, we developed a new task scheduling algorithm for the problem. In the experiments for evaluation, the proposed algorithm produced a schedule that is 36% faster than the compared methods. Since the proposed method makes the system finish execution of the tasks earlier, it also contributes for saving power consumption of the whole system. As a part of our future work, we are going to make our algorithm capable of accepting multiple task graphs in real time.

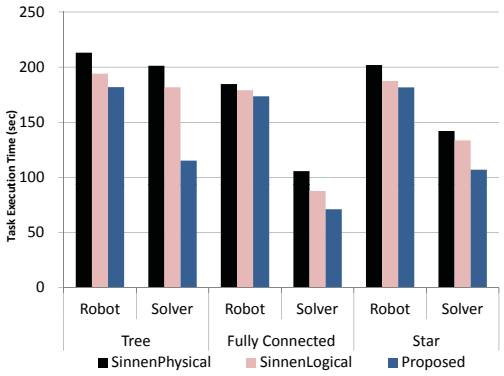


Figure 4: Simulation result

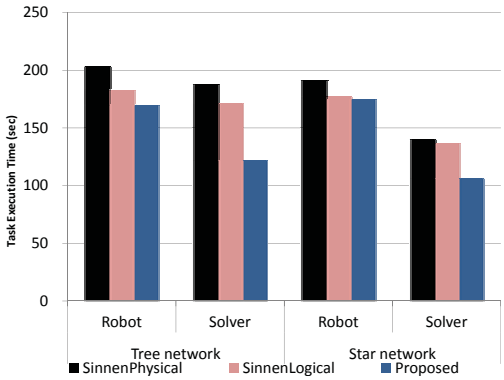


Figure 5: Results with real devices

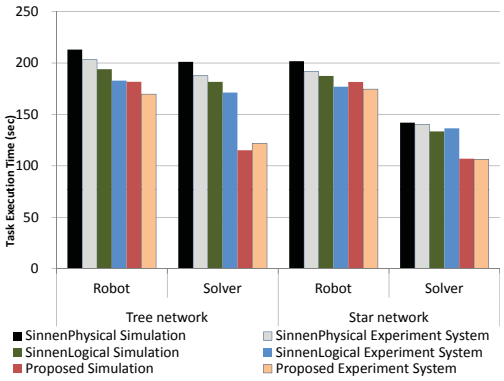


Figure 6: Comparison between estimated and real execution time of the whole task graph

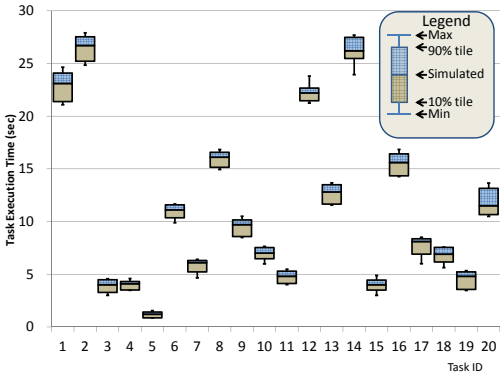


Figure 7: Comparison between estimated and real execution time of of each task node

REFERENCES

[1] Intel: Intel turbo boost technology in intel core microarchitecture(nehalem) based processors. Technical report, Intel (2008)

[2] Marr, D.T., Group, D.P., Corp, I.: Hyper-threading technology architecture and microarchitecture. Intel Technology Journal **6**(1) (2002) 4–15

[3] Kwok, Y., Ahamad, I.: Static scheduling algorithms for allocating directed task graphs to multprocessors. ACM Computing Surveys(CSUR) **31**(4) (dec 1999) 406–471

[4] Sinnen, O., To, A., Kaur, M.: Contention-aware scheduling with task duplication. Journal of Parallel and Distributed Computing **71**(1) (oct 2011) 77–86

[5] Sinnen, O., Sousa, L., Sandnes, F.: Toward a realistic task scheduling model. Parallel and Distributed Systems, IEEE Transactions on **17**(3) (mar 2006) 263–275

[6] Sinnen, O., Sousa, L.: Communication contention in task scheduling. Parallel and Distributed Systems, IEEE Transactions on **16**(6) (jun 2005) 503–515

[7] Gotoda, S., Ito, M., Shibata, N.: Task scheduling algorithm for multicore processor system for minimizing recovery time in case of single node fault. In: In Proceedings of Cluster, Cloud and Grid Computing (CCGrid), 12th IEEE/ACM International Symposium on. (may 2012) 260–267

[8] Park, J., Dally, W.J.: Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures. (2010) 1–10

[9] Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling **5**(5) (2002) 379–394

[10] Tobita, T., Kasahara, H.: Standard task graph set Home Page @ONELINE (2012) http://www.kasahara.elec.waseda.ac.jp/schedule/.